**What are HOCs and when should you use them?**

Let’s say that a user wants a component that increments a counter variable on every onClick event:

function ClickCounter() {

const [count, setCount] = useState(0); //default value of this state will be 0.

return (

<div>

{/\*When clicked, increment the value of 'count'\*/}

<button onClick={() => setCount((count) => count + 1)}>Increment</button>

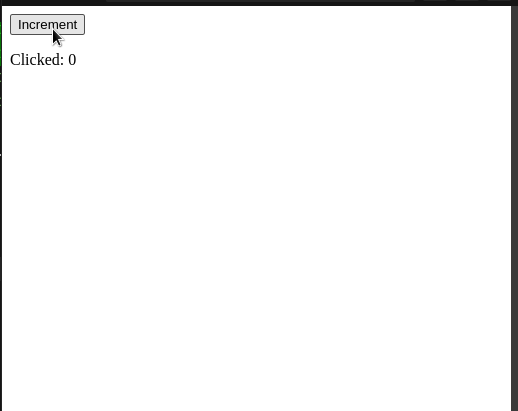
<p>Clicked: {count}</p> {/\*Render the value of count\*/}

</div>

);

}

export default ClickCounter;



Our code works! But what if the client wants another component that contains the same functionality, but it triggers on an onMouseOver event?

To make this possible, we would have to write the following code:

function HoverCounter(props) {

const [count, setCount] = useState(0);

return (

<div>

{/\*If the user hovers over this button, then increment 'count'\*/}

<button onMouseOver={() => setCount((count) => count + 1)}>

Increment

</button>

<p>

Clicked: {count}

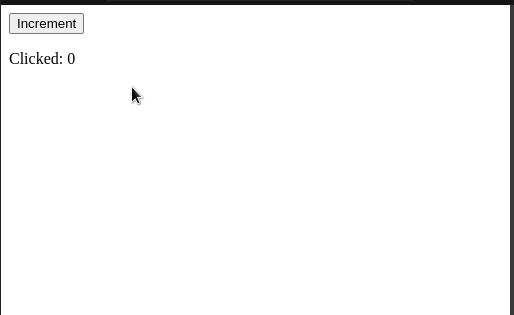
</p>

</div>

);

}

export default HoverCounter;



Even though our code samples are valid, there is a major problem: both of the files possess similar code logic. Consequently, this breaks the DRY concept. So how do we fix this issue?

This is where HOCs come in. Higher-order components allow developers to reuse code logic in their projects. As a result, this means less repetition and more optimized, readable code.

HOCs in React offer a versatile way to enhance the functionality and behavior of components. They can be applied to various use cases in your application. Here’s a list of common use cases for HOCs:

* **Conditional rendering**: Conditionally render components based on certain logic, such as user authentication or permission checks. A HOC can determine whether a component should be displayed and then wrap components with this HOC to make rendering decisions based on certain conditions
* **Authentication**: Implement user authentication and authorization. A HOC can protect routes or components, ensuring that only authenticated users have access. You can create an AuthHOC that checks the user’s authentication status and role. Wrap components or routes with this HOC to conditionally render contents based on user authentication and authorization
* **Data fetching**: Handle data fetching and loading states. A HOC can fetch data and pass it as props to the wrapped component, handling loading and error states
* **Styling**: Apply CSS styles or themes to components. A HOC can pass styling information as props to customize the appearance of components
* **State management**: Manage and share state, such as global app state or Redux store data, with multiple components using a HOC
* **Logging and analytics**: Implement logging, error tracking, or analytics by wrapping components with a HOC that reports events or errors
* **Caching and memoization**: Cache expensive computations or memoize functions to improve performance by using a HOC
* **Internationalization (i18n)**: Provide translation and internationalization features to components. A HOC can pass translated contents or language preferences

Remember that the flexibility of HOCs allows you to adapt them to various use cases in your React application, making your code more modular, reusable, and maintainable.

## The structure of a higher-order component

According to React’s documentation, a typical React HOC has the following definition:

*A higher-order component is a function that takes in a component and returns a new component.*

Using code, we can rewrite the above statement like so:

const newComponent = higherFunction(WrappedComponent);

In this line:

* newComponent: Will be the enhanced component
* higherFunction: As the name suggests, this function will enhance WrappedComponent
* WrappedComponent: The component whose functionality we want to extend. In other words, this will be the component that we want to enhance.

Here’s a detailed structure of a higher-order component in React:

First, create a function. Start by defining a JavaScript function that takes the base component as an argument and returns a new component with added functionality. In a functional HOC, you can use hooks for state and side effects:

import React, { useState, useEffect } from 'react';

const withEnhancement = (BaseComponent) => {

// HOC logic using hooks

return function EnhancedComponent(props) {

// HOC-specific logic using hooks

return (

<BaseComponent {...props} enhancedProp="someValue" />

);

};

};

Then, enhance the component. Inside the EnhancedComponent function, you can use hooks to manage state and perform side effects. Hooks like useState, useEffect, and useRef can be used to implement additional behavior:

const withEnhancement = (BaseComponent) => {

return function EnhancedComponent(props) {

const [count, setCount] = useState(0);

useEffect(() => {

// Perform side effects here

}, [count]);

return (

<BaseComponent {...props} enhancedProp="someValue" />

);

};

};

The next step is using the HOC. To use your functional HOC, wrap a component by passing it as an argument to your HOC function. The result will be a new component with the enhanced functionality:

const EnhancedComponent = withEnhancement(BaseComponent);

Using the enhanced component. You can use EnhancedComponent in your application just like any other React component, with the added functionality from the HOC:

function App() {

return (

<div>

<EnhancedComponent prop1="value1" prop2="value2" />

</div>

);

}

In the next segment of the article, we will see React’s HOC concept in action.

## Using higher-order components

### Initializing our repository

We have to first create a blank React project. To do so, start by writing the following code:

npx create-react-app hoc-tutorial

cd hoc-tutorial #navigate to the project folder.

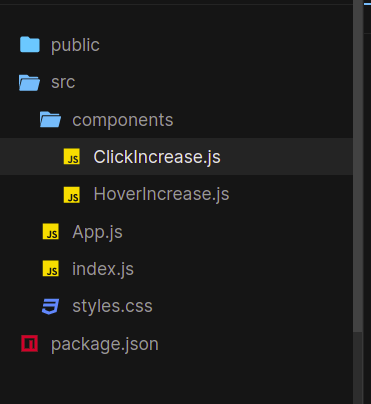
cd src #go to codebase

mkdir components #will hold all our custom components

For this article, we will build two custom components to demonstrate HOC usage:

* ClickIncrease.js: This component will render a button and a piece of text. When the user clicks on this button (an onClick event), the fontSize property of the text will increase
* HoverIncrease.js : This will be similar to that of ClickIncrease. However, unlike the former, this component will listen to onMouseOver events

In your project, navigate to the components folder. Here, create these two new files. When that’s done, your file structure should look like this:



Now that we have laid out the groundwork for the project, it’s time to build our custom components.

### Coding our components

In ClickIncrease.js, start by writing the following code:

//file name: components/ClickIncrease.js

function ClickIncrease() {

const [fontSize, setFontSize] = useState(10); //set initial value of Hook to 10.

return (

<div>

{/\*When clicked, increment the value of fontSize\*/}

<button onClick={() => setFontSize((size) => size + 1)}>

Increase with click

</button>

{/\*Set the font size of this text to the fontSize variable.\*/}

{/\*Furthermore, display its value as well.\*/}

<p style={{ fontSize }}>Size of font in onClick function: {fontSize}</p>

</div>

);

}

export default ClickIncrease;

Next, in your HoverIncrease component, paste these lines of code:

function HoverIncrease(props) {

const [fontSize, setFontSize] = useState(10);

return (

<div>

{/\*This time, instead of listening to clicks,\*/}

{/\*Listen to hover events instead\*/}

<button onMouseOver={() => setFontSize((size) => size + 1)}>

Increase on hover

</button>

<p style={{ fontSize }}>

Size of font in onMouseOver function: {fontSize}

</p>

</div>

);

}

export default HoverIncrease;

Finally, render these functions to the GUI like so:

//import both components

import ClickIncrease from "./components/ClickIncrease";

import HoverIncrease from "./components/HoverIncrease";

export default function App() {

return (

<div className="App">

{/\*Render both of these components to the UI \*/}

<ClickIncrease />

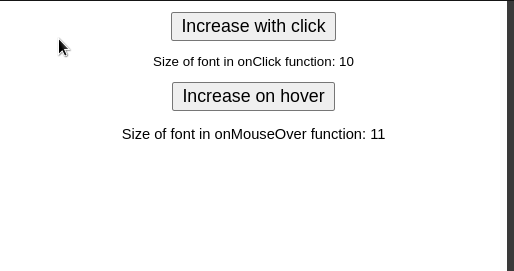
<HoverIncrease />

</div>

);

}

Let’s test it out! This will be the result of the code:



### **Creating and using our HOC function**

Within the components folder, create a file called withCounter.js. Here, start by writing the following code:

import React from "react";

const UpdatedComponent = (OriginalComponent) => {

function NewComponent(props) {

//render OriginalComponent and pass on its props.

return <OriginalComponent />;

}

return NewComponent;

};

export default UpdatedComponent;

Let’s deconstruct this code piece by piece. In the start, we created a function called UpdatedComponent that takes in an argument called OriginalComponent. In this case, the OriginalComponent will be the React element, which will be wrapped.

Then, we told React to render OriginalComponent to the UI. We will implement enhancement functionality later in this article.

When that’s done, it’s time to use the UpdatedComponent function in our app. To do so, first go to the HoverIncrease.js file and write the following lines:

import withCounter from "./withCounter.js" //import the withCounter function

//..further code ..

function HoverIncrease() {

//..further code

}

//replace your 'export' statement with:

export default withCounter(HoverIncrease);

//We have now converted HoverIncrease to an HOC function.

Next, do the same process with the ClickIncrease module:

//file name: components/ClickIncrease.js

import withCounter from "./withCounter";

function ClickIncrease() {

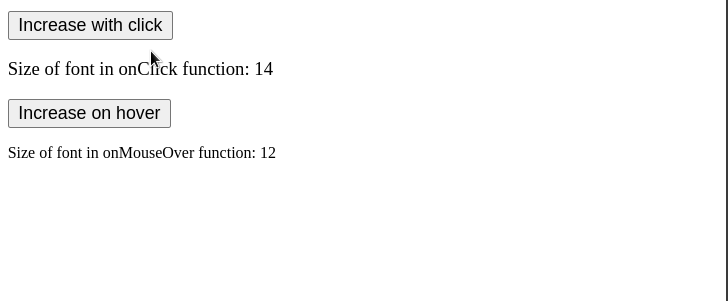
//...further code

}

export default withCounter(ClickIncrease);

//ClickIncrease is now a wrapped component of the withCounter method.

Let’s test it out! This will be the result in the code:



Notice that our result is unchanged. This is because we haven’t made changes to our HOC yet. In the next section, you will learn how to share props between our components.

### **Sharing props**

Using HOCs, React allows users to share props within the project’s wrapped components.

As a first step, create a name prop in withCounter.js like so:

//file name: components/withCounter.js

const UpdatedComponent = (OriginalComponent) => {

function NewComponent(props) {

//Here, add a 'name' prop and set its value of 'LogRocket'.

return <OriginalComponent name="LogRocket" />;

}

//..further code..

To read this data prop, all we have to do is make the following changes to its child components:

//extra code removed for brevity.

//In components/HoverIncrease

function HoverIncrease(props) { //get the shared props

return (

<div>

{/\* Further code..\*/}

{/\*Now render the value of the 'name' prop \*/ }

<p> Value of 'name' in HoverIncrease: {props.name}</p>

</div>

);

}

//Now In components/ClickIncrease.js

function ClickIncrease(props) {

//accept incoming props

return (

<div>

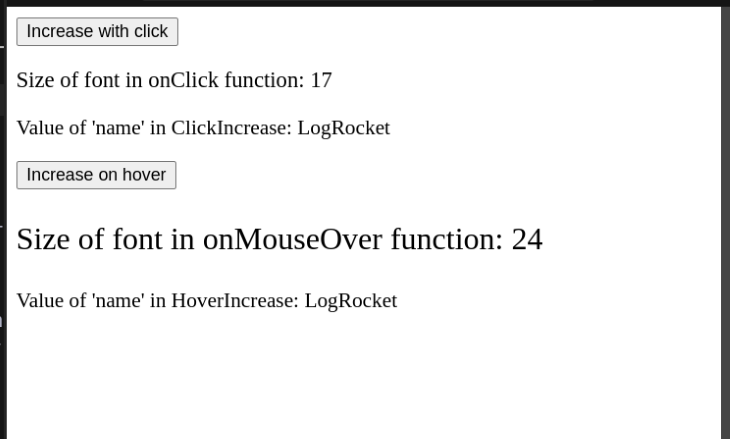
{/\*Further code..\*/}

<p>Value of 'name' in ClickIncrease: {props.name}</p>

</div>

);

}



That was easy! As you can see, React’s HOC design allows developers to share data between components with relative ease.

In upcoming sections, you will learn how to share states via HOC functions.

### **Sharing state variables with Hooks**

Just like with props, we can share Hooks using HOCs:

//In components/withCounter.js

const UpdatedComponent = (OriginalComponent) => {

function NewComponent(props) {

const [counter, setCounter] = useState(10); //create a Hook

return (

<OriginalComponent

counter={counter} //export our counter Hook

//now create an 'incrementSize' function

incrementCounter={() => setCounter((counter) => counter + 1)}

/>

);

}

//further code..

Here’s an explanation of the code:

* First, we created a Hook variable called counter and set its initial value to 10
* We also coded an incrementCounter function. When invoked, this method will increment the value of counter
* Finally, export the incrementSize method and the size Hook as props. As a result, this allows the wrapped components of UpdatedComponent to get access to these Hooks

As the last step, we now have to use the counter Hook. To do so, write these lines of code in the HoverIncrease and ClickIncrease module:

//make the following file changes to components/HoverIncrease.js and ClickIncrease.js

//extract the counter Hook and incrementCounter function from our HOC:

const { counter, incrementCounter } = props;

return (

<div>

{/\*Use the incrementCounter method to increment the 'counter' state..\*/}

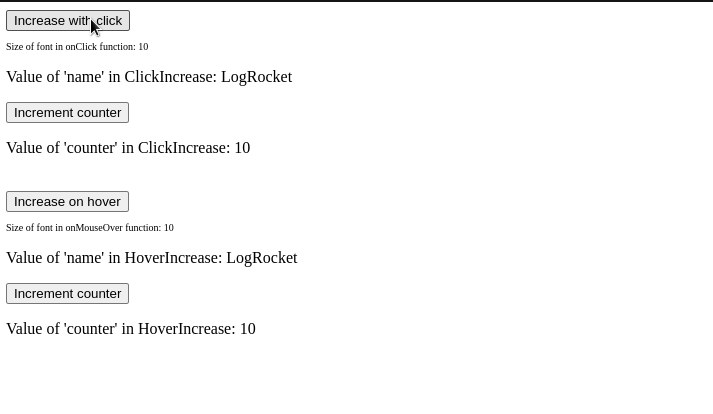
<button onClick={() => incrementCounter()}>Increment counter</button>

{/\*Render the value of our 'counter' variable:\*/}

<p> Value of 'counter' in HoverIncrease/ClickIncrease: {counter}</p>

</div>

);



Here, one important thing to notice is that the value of the counter state is not shared between our child components. If you want to share states between various React components, use React’s Context API, which allows you to effortlessly share states and Hooks throughout your app.

### **Passing parameters**

Even though our code works, consider the following situation: what if we want to increment the value of counter with a custom value? Via HOCs, we can even tell React to pass specific data to certain child components. This is made possible with parameters.

To enable support for parameters, write the following code in components/withCounter.js:

//This function will now accept an 'increaseCount' parameter.

const UpdatedComponent = (OriginalComponent, increaseCount) => {

function NewComponent(props) {

return (

<OriginalComponent

//this time, increment the 'size' variable by 'increaseCount'

incrementCounter={() => setCounter((size) => size + increaseCount)}

/>

);

//further code..

In this piece of code, we informed React that our function will now take in an additional parameter called increaseCount.

All that’s left for us is to use this parameter in our wrapped components. To do so, add this line of code in HoverIncrease.js and ClickIncrease.js:

//In HoverIncrease, change the 'export' statement:

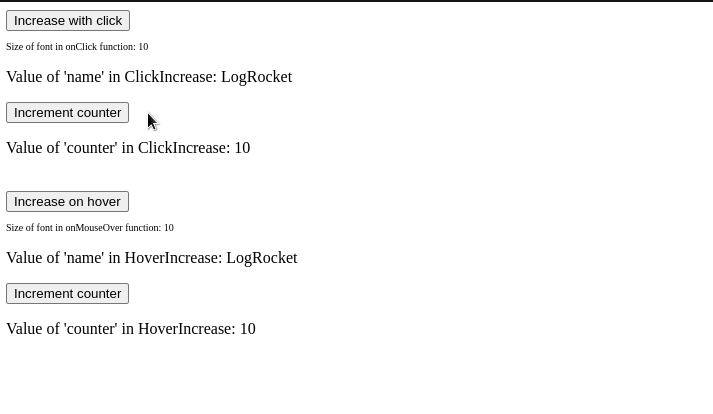
export default withCounter(HoverIncrease, 10); //value of increaseCount is 10.

//this will increment the 'counter' Hook by 10.

//In ClickIncrease:

export default withCounter(ClickIncrease, 3); //value of increaseCount is 3.

//will increment the 'counter' state by 3 steps.



In the end, the withCounter.js file should look like this:

import React from "react";

import { useState } from "react";

const UpdatedComponent = (OriginalComponent, increaseCount) => {

function NewComponent(props) {

const [counter, setCounter] = useState(10);

return (

<OriginalComponent

name="LogRocket"

counter={counter}

incrementCounter={() => setCounter((size) => size + increaseCount)}

/>

);

}

return NewComponent;

};

export default UpdatedComponent;

HoverIncrease.js should look like this:

import { useState } from "react";

import withCounter from "./withCounter";

function HoverIncrease(props) {

const [fontSize, setFontSize] = useState(10);

const { counter, incrementCounter } = props;

return (

<div>

<button onMouseOver={() => setFontSize((size) => size + 1)}>

Increase on hover

</button>

<p style={{ fontSize }}>

Size of font in onMouseOver function: {fontSize}

</p>

<p> Value of 'name' in HoverIncrease: {props.name}</p>

<button onClick={() => incrementCounter()}>Increment counter</button>

<p> Value of 'counter' in HoverIncrease: {counter}</p>

</div>

);

}

export default withCounter(HoverIncrease, 10);

And finally, your ClickIncrease component should have the following code:

import { useEffect, useState } from "react";

import withCounter from "./withCounter";

function ClickIncrease(props) {

const { counter, incrementCounter } = props;

const [fontSize, setFontSize] = useState(10);

return (

<div>

<button onClick={() => setFontSize((size) => size + 1)}>

Increase with click

</button>

<p style={{ fontSize }}>Size of font in onClick function: {fontSize}</p>

<p>Value of 'name' in ClickIncrease: {props.name}</p>

<button onClick={() => incrementCounter()}>Increment counter</button>

<p> Value of 'counter' in ClickIncrease: {counter}</p>

</div>

);

}

export default withCounter(ClickIncrease, 3);